Relational Compilation for Performance-Critical Applications

Extensible Proof-Producing Translation of Functional Models into Low-Level Code

Clément Pit-Claudel*
EPFL and Amazon AWS
Lausanne, Switzerland

Jade Philipoom†
MIT CSAIL
Cambridge, MA, USA

Dustin Jamner
MIT CSAIL
Cambridge, MA, USA

Andres Erbsen
MIT CSAIL
Cambridge, MA, USA

Adam Chlipala
MIT CSAIL
Cambridge, MA, USA

Abstract

There are typically two ways to compile and run a purely functional program verified using an interactive theorem prover (ITP): automatically extracting it to a similar language (typically an unverified process, like Coq to OCaml) or manually proving it equivalent to a lower-level reimplementation (like a C program). Traditionally, only the latter produced both excellent performance and end-to-end proofs.

This paper shows how to recast program extraction as a proof-search problem to automatically derive correct-by-construction, high-performance code from purely functional programs. We call this idea *relational compilation* — it extends recent developments with novel solutions to loop-invariant inference and genericity in kinds of side effects.

Crucially, relational compilers are incomplete, and unlike traditional compilers, they generate good code not because of a fixed set of clever built-in optimizations but because they allow experts to plug in domain–specific extensions that give them complete control over the compiler's output.

We demonstrate the benefits of this approach with Rupicola, a new compiler-construction toolkit designed to extract fast, verified, idiomatic low-level code from annotated functional models. Using case studies and performance benchmarks, we show that it is extensible with minimal effort and that it achieves performance on par with that of handwritten C programs.

---

*All work was done prior to joining Amazon.
†Jade Philipoom is now at Google.

CCS Concepts: • **Software and its engineering → Compilers**; **Software verification**.

Keywords: compilation, verification, theorem proving

1 Introduction

Vulnerabilities in critical systems fall into roughly two categories: logic mistakes (incorrect business logic) and programming mistakes (use-after-free, out-of-bounds accesses). High-level languages attempt to eliminate both: logic mistakes by promoting higher levels of abstraction that facilitate reasoning about program behavior, and low-level issues through safer programming paradigms (garbage collection to rule out use-after-free errors, stream- and result-oriented APIs for out-of-bounds accesses, etc.).

At the extreme, *purely functional* languages offer very strong protections against low-level mistakes and readily lend themselves to mathematical reasoning. By eliminating mutable arrays, exceptions, state, and other low-level concerns and encouraging higher-order programming, languages like Coq [50], Lean [5], Idris [3], or the pure subsets of Haskell and F* [12, 48] offer programming models much less susceptible to the low-level issues that plague the vast majority of today's critical systems.

Unfortunately, such a combination of flexibility and safety comes at a significant performance cost: it is an unsolved problem to program a compiler for any of these purely functional languages that verifiably preserves all of their high-level guarantees while offering performance competitive with the usual low-level suspects, especially C (Box 1).

As a result, most critical systems are written in low-level stateful languages, and verification efforts have to deal head-first with the complexity of low-level programming — ambient effects like state, explicit memory management, complexities of memory layout, etc.

In this paper we attempt to chart a different course, leveraging code-generating proof search to construct compositional, extensible domain-specific compilers that perform advanced domain- or even program-specific transformations in a safe manner. We start with a comprehensive treatment of relational compilation, a program-derivation technique that soundly bridges the gap from shallowly embedded programs to deeply embedded executable code, with special emphasis on composability and extensibility.

We then provide a real-world perspective on relational compilation, using it to derive, in the Coq proof assistant, high-performance implementations of various small yet bug-prone low-level programs. Because our focus is on low-level programming, we do not attempt to compile *all* or even *most* functional programs. Instead, we focus on small, loop-heavy programs such as those often found in binary parsers, text-manipulation libraries, cryptographic routines, system libraries, and other high-risk, high-performance code. These programs are not traditionally implemented in purely functional languages, but we show that relational compilation can in fact be used to write performance-critical programs in that style, directly within the native, pure logic of an interactive theorem prover, combining straightforward reasoning and verification with excellent performance. In other words, our programs are pure and written with maps and folds, but they compile to code that manages its own memory, mutates its inputs, and runs at the speed of vectorizable for loops.

Rupicola, our compiler-construction toolkit, is not a general-purpose compiler and is not intended to replace all program extraction. Instead, Rupicola is restricted, out of the box, to a minimal set of constructs (essentially arithmetic, simple data structures, and some control flow), yielding a predictable and transparent compilation process. Users are expected (and enabled) to extend it as needed for each new domain, plugging in domain- or program-specific compilation hints that capture the insight that humans would normally apply when manually implementing high-level specifications in a low-level language: details of memory layout and memory management, implementation strategies for data-structure traversals, etc.[1]

In a sense, Rupicola codifies and automates away the most unpleasant part of the traditional end-to-end verification pipeline: connecting handwritten low-level code to its functional specification. In the traditional world, authors not willing to rely on Coq's extraction (for performance or trust

---

[1]Our implementation is free software, available under the MIT (Expat) license at https://github.com/mit-plv/rupicola/. All benchmarks and code samples in this paper can be downloaded as part of a preprovisioned virtual machine at https://zenodo.org/record/6330612 [40].

To ground this discussion of high-level inefficiencies, consider the simple task of converting an ASCII string to uppercase.

In a purely functional language like Gallina (Coq), it may succinctly be written as follows (with strings being linked lists of characters, characters an inductive type with 256 cases, and toupper a disjunction with one case per lowercase letter plus an unchanged default):

```
String.map Char.toupper str
```

This program accurately captures the intent of the task, but how fast does it run? When extracted to OCaml, it will pointer-chase through a linked list to traverse the original string (creating data dependencies and cache pressure), create a fresh string (costing allocations, cache misses, and an extra traversal for garbage collection), and either stack-overflow on long strings (due to a non-tail-recursive map, though there have been recent developments in that space [10]), or traverse the string twice (doubling allocation and pointer-chasing costs), or accumulate continuations (even more allocations).

Assuming that the original string is never reused, the C implementation below performs a single pass, occupies constant stack space, does not allocate, is cache-friendly, can be unrolled, and is trivially vectorizable (toupper on ASCII chars is just a comparison and a bitmask). The purely functional version doesn't stand a chance.

```
for (int i = 0; i < len; i++)
  str[i] = toupper(str[i]);
```

Box 1. Compiler inefficiencies.

reasons) will manually relate handwritten, deeply embedded low-level programs to functional models, and then they will separately relate each functional model to a high-level specification. In that world, authors must repeatedly deal with the complexities of the low-level language's semantics and with details such as when to allocate or free memory or how to relate low-level memory layouts to high-level functional models. Rupicola, in contrast, completely automates the first phase of this process, generating the deeply embedded low-level program from its functional model by leveraging user-provided hints and program annotations. In Rupicola, programmers only supply *shallowly* embedded programs written in a subset of Gallina that naturally maps to low-level constructs, and the tooling produces low-level, deeply embedded code. Unchanged is the second phase that relates these functional models to abstract specifications: that part is still the programmer's responsibility. But because Rupicola's inputs are shallowly embedded, this phase is disconnected from the details of the low-level language's semantics, and the traditional reasoning patterns best supported by Coq — especially structural induction — are fully applicable (Figure 1).
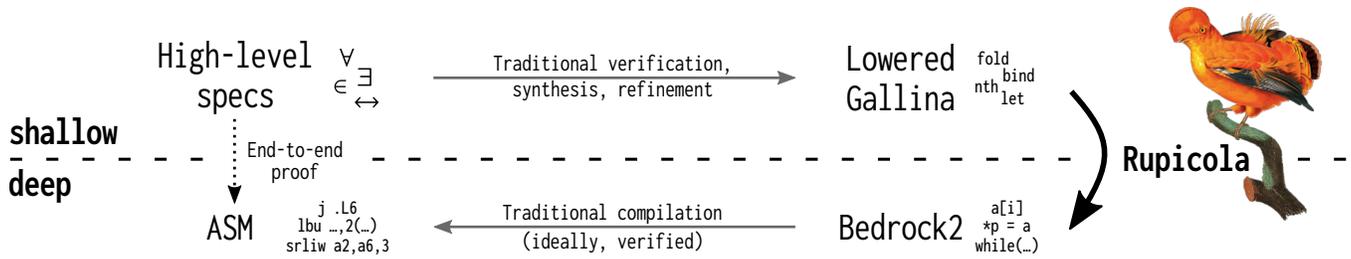
Figure 1. Where Rupicola fits in the bigger picture. Rupicola is not a universal compiler: it bridges the shallow-to-deep gap by accepting a restricted (but extensible) input that can reliably and predictably be translated to fast low-level code.

The main contributions of this paper are its presentation of a flavor of proof-producing code generation that we call *relational compilation*, and its application of that technique to the automatic generation of verified, high-performance low-level code:

- We develop a novel, systematic presentation of code generation via proof search, starting from a traditional verified compiler and progressively transforming it to introduce relational compilation, with a focus on composability and extensibility.
- We describe and evaluate Rupicola, a relational compiler from Gallina, Coq's programming language, to Bedrock2 [8], a low-level imperative language. Rupicola advances the state of the art through composable support for arbitrary monadic programs, novel treatment of loops, and output-code performance.

There have been many previous efforts in this space, foremost among them developments on Imperative/HOL [17, 18], CakeML [13, 30], Œuf [15, 27], HOL compilation to Verilog [23], Fiat-to-Facade [41], CertiCoq [1], and Low* [45]. We provide detailed comparisons in section 5. Briefly, **Rupicola's novelty is its combination of performance, foundational proofs, and extensibility**:

- All projects above except CertiCoq and Low* use proof-producing code-generation. Among these, only Fiat-to-Facade focuses on generating high-performance low-level code from functional programs, but its performance does not match that of Rupicola (others target either garbage-collected languages or other types of languages like Verilog; LLVM/HOL for example compiles from a one-to-one shallow embedding of LLVM).
- KreMLin, Low*'s compiler, does produce code with performance matching handwritten programs, but it is not formally verified (Rupicola's output is certified by a proof of total correctness). CertiCoq has proofs (though not for the initial reification step), but it is a standard compiler whose outputs require a runtime system.
- Fiat-to-Facade supports straightforward user extensions, but unlike in Rupicola users are limited by the linearity of the target language, and support for loops and effects is

ad-hoc (loops can mutate only one object, and the nondeterminism monad is hardcoded).

We call the output of Rupicola *idiomatic* not because it is particularly readable (Rupicola uses Bedrock2's pretty-printer to C) but because it employs the constructs and patterns that a programmer would use in handwritten C code (unlike the output of a regular compiler from Gallina to C, which would encode in C the high-level patterns found in the source).

2 On Relational Compilation

The traditional process for developing a verified compiler is to define types that model the source (S) and target (T) languages, give these language *semantics*, and write a function $f : S \rightarrow T$ proven to preserve semantics. That is, if $\sigma_S(s)$ denotes the semantics of $s \in S$ and $\sigma_T(t)$ those of $t \in T$, then f is correct iff $f(s) \sim s$ for all s, where $t \sim s$ iff $\sigma_T(t) = \sigma_S(s)$.

It turns out that instead of writing the compiler as a monolithic program and separately verifying it, we can break up the compiler and its proof into a collection of orthogonal correctness theorems and *use these theorems* to drive a code-generating proof-search process — a process we call *relational compilation*. Instead of functions $f : S \rightarrow T$, we implement compilers as automated decision procedures for theorems of the form $\exists t, t \sim s$. In a constructive setting, any proof of that statement must exhibit a witness t, which will be a (correct) compiled version of s.[2]

The two main benefits of relational compilation are flexibility and trustworthiness: it provides a very natural and modular way to think of compiler extensions and makes it possible to extract shallowly embedded programs without trusting an extraction routine (in contrast, extraction in Coq is trusted). The main cost? Completeness: a (total) function always terminates and produces a compiled output; a (partial) proof-search process may loop or fail.

The following presentation of relational compilation inherits from a long research tradition: code-generating proof

---

[2]The theorem does not ∀-quantify s because we want to generate one distinct proof *per input program* — otherwise, with a ∀s quantification, the theorem would be equivalent by Skolemization to defining a single compilation function f, which we are trying to avoid.

search has been referred to as *proof-producing compilation, synthesis*, or *translation*, and, when the source language is shallowly embedded, *proof-producing extraction, certifying extraction*, or *binary code extraction*. [15, 29, 31, 41]

2.1 A Step-by-Step Example in Coq

Here is a concrete pair of languages that we will use as a demonstration. Language S is a simple arithmetic expression language. Language T is a stack machine.

Language definitions. On the first line below, in Coq syntax, is a language S with constants and addition. Programs in language T are lists (on the third line) of stack operations (on the second line), either pushing a constant or popping two values from the stack and pushing their sum.

```
Inductive S := SInt z | SAdd (s1 s2 : S).
Inductive T_Op := TPush z | TPopAdd.
Definition T := list T_Op.
```

Semantics. Interpreters for these languages are easy to define by recursion, mapping the constructors of S and T_Op to operations on $\mathbb{Z}$. This, in turn, is enough to formulate a (contextual) equivalence relation on S and T:

```
Fixpoint σS (s: S) :=
  match s with
  | SInt z     ⇒ z
  | SAdd s1 s2 ⇒ σS s1 + σS s2 end.
Definition σOp (zs: list Z) (op: T_Op) :=
  match op, zs with
  | TPush z, zs           ⇒ z      :: zs
  | TPopAdd, z2::z1::zs ⇒ z1+z2 :: zs
  | _, zs ⇒ zs (* Invalid: no-op *) end.
Definition σT (t: T) (zs: list Z) :=
  List.fold_left σOp t zs.
Notation "t ~ s" :=
  (∀ zs, σT t zs = σS s :: zs).
```

Compilation. Here is a single-pass compiler for our pair of languages. The SInt case maps to a stack-machine program that simply pushes the constant z on the stack, and the SAdd case produces a program that pushes both operands in succession before computing their sum using the TPopAdd opcode. Its proof is straightforward by induction.

```
Fixpoint StoT (s: S) := match s with
 | SInt z     ⇒ [TPush z]
 | SAdd s1 s2 ⇒ StoT s1 ++ StoT s2 ++ [TPopAdd]
end.
Lemma StoT_ok : ∀ s, StoT s ~ s. Proof. ... Qed.
```

2.2 Compiling with Relations

Like all functions, StoT can be rewritten as a relation[3]; here is one way to do so (each constructor corresponds to a branch in

the original recursion, and each x $\mathcal{R}$ y premise corresponds to a recursive call to StoT):

```
Inductive StoT_rel : T → S → Prop :=
| StoT_RInt : ∀ z, [TPush z] R SInt z
| StoT_RAdd : ∀ t1 s1 t2 s2,
    t1 R s1 → t2 R s2 →
    t1 ++ t2 ++ [TPopAdd] R SAdd s1 s2
where "t 'R' s" := (StoT_rel t s).
```

Compiler correctness for the relation $\mathcal{R}$ reduces to inclusion: $\mathcal{R}$ *defines a* correct *mapping from S to T (possibly one-to-many, possibly suboptimal, but* correct*) iff its graph is a subset of the graph of* ~ (with unchanged proof structure):

```
Theorem StoT_rel_ok : ∀ t s, t R s → t ~ s.
```

Naturally we can use $\mathcal{R}$ to prove specific program equivalences, but more importantly we can use it to *run* the compiler, with proof search! To *compile* s, we simply search for a program t such that t $\mathcal{R}$ s[4].

```
Example s7 := SAdd (SInt 3) (SInt 4).
Example t7_rel: { t7 | t7 R s7 }.
  unfold s7; eexists.
```

```
?t7 R SAdd (SInt 3) (SInt 4)
```

The value **?t7** is a placeholder (an existential variable, or *evar*) corresponding to the program that we are deriving, which we can *refine* as a side effect of applying a lemma. After applying the lemma StoT_RAdd, Coq asks us to provide two subprograms, each corresponding to one operand of SAdd.

```
  apply StoT_RAdd.
```

?t1 R SInt 3	?t2 R SInt 4

```
  all: apply StoT_RInt. Defined.
Compute t7_rel.
```

```
= exist [TPush 3; TPush 4; TPopAdd]
```

We can also use Coq's inspection facilities to see the proof term as it is being generated (this time the boxes show the proof term, not the goals): each lemma application is equivalent to a recursive call in a run of StoT.

```
(exist ?t7)
```

```
  apply StoT_RAdd.
```
```
(exist (?t1 ++ ?t2 ++ [TPopAdd]))
```

```
  all: apply StoT_RInt.
```
```
(exist ([TPush 3] ++ [TPush 4] ++ [TPopAdd]))
```

This is traditional *logic programming*, applied to compilers. Reemphasizing the key insight: *correctly compiling a program s is the same as proving* $\exists t, t \sim s$.

---

[3]Any function $f : x \mapsto f(x)$ defines a relation $t \sim_f s$ iff $t = f(s)$, sometimes called the graph of f.

[4]For the purpose of this document, { t | P } can be considered a synonym for $\exists$ t, P.

2.3 Open-Ended Compilation

The proofs of correctness of StoT and $\mathfrak{R}$ have the exact same structure: each is made up of two orthogonal lemmas.

```
Lemma StoT_SInt z : [TPush z] ~ SInt z.

Lemma StoT_Plus t1 s1 t2 s2 :
  t1 ~ s1 → t2 ~ s2 →
  t1 ++ t2 ++ [TPopAdd] ~ SAdd s1 s2.
```

Each of these is really a standalone fact, and each corresponds to a *partial* relation between S and T (partial because each lemma is only applicable to *some* pairs of input/output programs in S and T, like TPopAdd and SAdd in StoT_Plus). In other words: *A relational compiler is just a collection of facts connecting target programs to source programs.*

As a result, we do not even need to define a relation: we can prove these lemmas directly, and use them to build a compiler — just like we did the constructors of $\mathfrak{R}$! Coq has facilities ("hint databases") to perform automatic proof search using a set of lemmas, which we can use to automate compilation.[5]

Relational compilation offers a crucial benefit: *composability*. This is particularly useful when compiling (shallowly) embedded domain-specific languages (EDSLs), especially when the compiler needs to be extensible.[6]

2.4 Compiling Shallowly Embedded DSLs

The original setup of the problem required us to exhibit a function $f : S \to T$. Not so with relational compilation, which instead requires us to prove instances of the $\sim$ relation (one per program). Thus we can apply this compilation technique to compile *shallowly* embedded DSLs[7].

To change our running example to compile arithmetic expressions written directly in Gallina, we start by redefining the relation to use *Gallina expressions* on the right side of the equivalence (there is no longer a reference to S and σS) and then add compilation lemmas referring to plain-Gallina operations (eg. +, not SAdd) so that each now relates a shallow program to an equivalent deeply embedded one:

```
Notation "t ≈ z" := (∀ zs, σT t zs = z :: zs).
Lemma GallinatoT_Z z : [TPush z] ≈ z.
Lemma GallinatoT_Zadd t1 z1 t2 z2 :
  t1 ≈ z1 → t2 ≈ z2 →
  t1 ++ t2 ++ [TPopAdd] ≈ z1 + z2.
```

These lemmas are sufficient to assemble a compiler: once we populate a hint database with compilation lemmas, we can (relationally) compile shallowly embedded programs:

```
Example t7_shallow: { t7 | t7 ≈ 3 + 4 }.
Proof. typeclasses eauto. Defined.
Compute t7_shallow.
```

```
= exist [TPush 3; TPush 4; TPopAdd]
```

There is something slightly magical happening here. By rephrasing compilation as a proof-search problem, we have managed to make a compiler that would not even be expressible (let alone provable!) as a regular Gallina function. Reasoning on shallowly embedded programs is often much nicer than reasoning on deeply embedded programs, and this technique offers a convenient way to bridge the gap.

3 Real-World Relational Compilation

The first part of this paper presented the key ideas behind relational compilation, keeping implementation details to a minimum. In this part we discuss how these ideas come together to implement a realistic compiler-construction toolkit. Specifically, we present the design and implementation of *Rupicola*, a compiler-construction framework with a focus on simple, low-level performance-critical programs.

Rupicola allows users to build compilers that extract shallowly embedded programs written in subsets of Gallina (the functional language of the Coq proof assistant), such that every successful compilation yields a deeply embedded program in Bedrock2 (a C-like language [8] that can be compiled to RISC-V or pretty-printed to C, see Box 2), along with a proof of equivalence to the original functional program.[8]

Rupicola is implemented in Coq, using a mix of Coq lemmas (to relate high-level functional code patterns to low-level imperative ones) and Ltac tactics (to guide the application of these lemmas). Its core is very small (hundreds of lines), but thanks to a variety of extensions we end up with a reasonably expressive input language: with all extensions loaded, we have support for arithmetic over many types (Booleans, bounded and unbounded natural numbers, bytes, integers, machine words), various control-flow patterns (conditionals as well as iteration patterns like maps and folds, with and without early exits), various flat data structures such as mutable cells and arrays; plain and monadic binds; various monadic extensions including the nondeterminism, state, writer, and I/O monads and a generic free monad; and various low-level effects and features such as stack allocation, inline tables, intrinsics, and external functional calls.

We designed Rupicola so that the default reaction to unexpected input is to stop and ask for user guidance, rather than fall back to a slower generic implementation. As a result, Rupicola makes few guesses (in contrast to more clever

---

[5]Hint databases provide a convenient way to package lemmas together, but the key to extensibility is really the idea of phrasing compilation as proof search, as it enables *sound composition*.

[6]Most compilers restrict user extensions to single-language AST transformations: the passes that translate between intermediate languages are fixed, monolithic functions. Relational compilation, in contrast, offers a flexible way to customize cross-language translators — so much so that relational compilers are built by composing independent mini-compilers that each support just one language construct.

[7]A shallow embedding is one where programs are defined directly in the host language, in contrast with a deep embedding where programs are represented as abstract syntax trees, i.e. data.

[8]Our Coq-to-Bedrock compilation toolkit is named after *Rupicola Rupicola*, the Guianan Cock-of-the-Rock, depicted in Figure 1.

approaches, e.g. superoptimization [26, 38]) — and consequently very few *incorrect* guesses. Programs compiled using Rupicola achieve performance indistinguishable from that of handwritten C programs because they *are* (semantically) indistinguishable from handwritten C programs.

We start by presenting Rupicola's philosophy and design goals and giving a flavor of using it by stepping through the compilation of a complete example. Then, we explain the core architecture of Rupicola and highlight two particular challenges: genericity in computational effects and loop-invariant inference.

3.1 Design goals

Rupicola's goal is to enable programmers to generate fast low-level code while reasoning about nice and simple high-level functional programs. It is designed to offer developers of critical software (programmers who want both high performance and end-to-end proofs) a better solution than the traditional approach of writing low-level code directly and then separately verifying it against a high-level specification.

These (manual) proofs relating low-level imperative code to purely functional specifications are one of the main pain points of that traditional approach: they mix high-level reasoning about domain logic with low-level details of control flow, state, and memory management, making them complicated and tedious.

With Rupicola, programmers do not write low-level programs by hand. Instead, a programmer starts with a high-level specification (typically a function, a relation, or a set of properties that the program should have), implements a functional model of that specification (a purely functional program written in a subset of Gallina, optionally adorned with performance annotations), specifies a binary interface (an ABI, the collection of low-level representation choices that are visible to other low-level code but abstracted-away in the high-level code), and finally uses relational compilation (that is, code-generating proof search) to generate imperative code: the shape of the model, combined with the lemmas introduced into the relational compiler, determines what code gets generated.[9]

The first part of this process (constructing and verifying a functional model) is done by hand, but the process is straightforward, and the proof of correctness is very easy: the logic of a proof assistant like Coq is very pleasant to work with when reasoning about purely functional, shallowly embedded programs.

The rest of this process (relational compilation) is where most of the work happens. Instead of writing imperative code by hand, experts capture and prove (once and for all)

the tricks that they would have used when transforming a functional prototype into an efficient low-level implementation. These performance insights can then be utilized to automatically *generate* low-level code, along with proofs of correctness, in the same way that we used lemmas in section 2 to generate stack-machine programs.

When a user attempts to compile a program that uses unsupported constructs, or a program whose compilation requires solving side conditions that Rupicola's logic does not recognize (e.g. tricky side conditions on array bounds or integer overflows), Rupicola makes as much progress as possible and then presents unsolved compilation subgoals to the user, who may then plug in new lemmas to implement missing constructs, or new tactics to discharge unsolved side conditions (we evaluate the cost of developing these extensions in section 4). This means that users never have to guess at what is happening: they can learn the shape of missing lemmas from the goals printed by Rupicola.

Letting users construct their own domain-specific compilers means that Rupicola is *predictable*: compilers built with Rupicola (almost) never backtrack and do not second-guess the user by introducing potentially unwanted transformations — there are no performance surprises. Instead, Rupicola generates good code *by leveraging the insight that the user provides*. This is why extensibility is so important: it enables users to do all their programming and proving in the comfortable confines of a purely functional language, yet get complete control over the extraction of that code — including algorithms, data layout, and performance.

3.2 Compiling with Rupicola

Let us revisit the example from Box 1, with Rupicola. The source program is upstr := (λ s $\Rightarrow$ String.map toupper s). Gallina's string type is a linked list of characters, each represented as an 8-tuple of Booleans; String.map is a simple recursive function; and toupper is specified as a match mapping each lowercase ASCII character to its uppercase counterpart: "a" $\Rightarrow$ "A", "b" $\Rightarrow$ "B", etc. The target program is a for loop that mutates an array, which differs from the source program in four ways:

1. First, we change to a more compact representation of strings (C's char* type), to save memory and improve cache locality.
2. Second, we eliminate higher-order iteration by replacing the recursive map with a loop, to prevent stack overflows.
3. Third, we change to in-place mutation, to eliminate memory allocations and reduce cache pressure.
4. Fourth, we introduce bit tricks specific to ASCII in the computation of uppercase characters, to save time.

Because of the simplicity of this example, the lowering from the high-level specification is almost trivial. We just define a variant upstr' of upstr on the type list byte instead of string, using the ListArray.map iterator

---

[9]Ignoring the fact that we automatically generate the low-level code instead of asking the user to write it, this problem decomposition is rather typical: just like in tools like Low* [45], Fiat [6], or VST [2], it lets us reason about algorithmic changes at the functional level and separately handle the jump to a low-level language.

instead of `String.map`: `upstr' := (λ s ⇒ let/n s := ListArray.map (λ b ⇒ a2b (toupper (b2a b))) s in s)`.

Proving this model equivalent to the original function is a matter of three lines, but it contains almost all that we need to guide the compiler (as we will see, the `let`/n-binding tells Rupicola to mutate the variable named s, and `ListArray.map` hints at the in-memory representation of the string). And with that, we are ready to start!

The first transformation (strings as arrays, chars as bytes) we encode as part of the ABI of our low-level program: a pre-postcondition pair. We state that we start with a pointer p to a buffer containing the same data as the string in a byte array, and that we end with the same memory containing the same string transformed to uppercase. The corresponding specification is shown below; it says how the low-level program that we intend to generate should be called and what it will return:

```
Instance spec : spec_of "upstr" :=
  fnspec! "upstr" p wlen / s r, {
    requires tr m := wlen = of_nat (length s) ∧
      (array p s * r) m;
    ensures tr' m' := tr' = tr ∧
      (array p (upstr' s) * r) m' }.
```

Here is how to read this function specification (`fnspec`). The function takes two arguments (machine words) p (a pointer) and wlen and two ghost arguments s (a list of bytes) and r (a separation-logic predicate), and it returns nothing; the `requires` clause specifies how the function is called (with a condition on argument wlen and on the memory m[10]); and the `ensures` clause states that the program does not produce observable I/O (`tr' = tr`) and that it writes the updated string (`upstr' s`). Each separation-logic predicate has an `array` fact that takes a pointer p and a Gallina-level list, plus the separated frame r modeling the rest of the memory. Predicates for arrays are used throughout Rupicola, so we do not have to define anything new for this part.

The second transformation (map as a loop) is done using a compiler extension — a lemma translating `ListArray.map` into a `for` loop. This sort of translation is a common pattern, so Rupicola's standard library has built-in support for it; we just need to load the relevant library and plug in Coq's linear-arithmetic solver to handle index-bounds side conditions.

The third transformation (mutation) comes as a side effect of using an in-place map-to-loop lemma. Its application is guided by the use of `let`/n ("let/named"), which is like a regular `let` but annotated with a variable name. In Rupicola we call this an *intensional* mutation effect, since it is introduced automatically by analyzing the source code (and not explicitly encoded using a state monad).

---

[10]All examples in this paper use nonoverlapping inputs, but this rule is not a restriction of Rupicola: the compiler also supports reasoning about nonseparating conjunctions, which are useful for programs like `memmove`, which allows its source and destination to overlap.

Rupicola compiles to Bedrock2 [8], an untyped version of the C programming language. It has a verified compiler to RISC-V with a complete correctness proof as well as a minimal program logic. The semantics divide the program state in three parts: the heap (a flat array of bytes indexed by natural numbers, with an optional layer of separation logic in the program logic), the current function context (a map of names to machine words), and an event trace capturing externally observable events.

Bedrock2's structured control flow includes function calls, conditionals, and loops; the semantics only give meaning to terminating loops, so proofs about Bedrock2 programs are total-correctness proofs. Additionally, stack usage is measured and restricted, so there is no general recursion. Memory allocation is handled by client code, except for allocation on the stack, which is available through a language primitive that gives client code access to temporary scratch space that is lexically scoped within a function's body.

Box 2. Rupicola's target language: Bedrock2

The last transformation, efficient uppercasing, can be plugged into the compiler as a rewrite. First we prove a program equivalence between our `toupper` function on 8-tuples of Booleans and an efficient byte computation:

```
Definition toupper' (b: byte) : byte :=
  if wrap (b - "a") <? 26 then b & x5f else b.
```

Then we plug it in as a hint that applies inside the body of the function passed to `ListArray.map`, along with an unfolding hint that allows Rupicola to inline the function `toupper'`. Once all these pieces are together we can invoke the compiler, and we get the expected low-level program with no further manual intervention beyond loading the appropriate compiler submodules, which pulls in the 20 or so lemmas that are needed to handle this example completely:

```
Derive upstr_br2fn SuchThat
    (defn! "upstr" ("s", "len") { upstr_br2fn },
     implements upstr') As upstr_br2fn_ok.
Proof. compile. Qed.
```

The result is a Bedrock2 program `upstr_br2fn` and its proof of (total) correctness `upstr_br2fn_ok` (the `Derive` command is syntactic sugar for defining a dependent pair). The `defn!` part specifies which function we are compiling (`upstr'`), its spec (`"upstr"` matches the `spec_of` instance above), and its signature (`"s"` and `"len"`).

Finally, the program can be further compiled using Bedrock2's verified compiler (with support for linking against separately compiled (or handwritten) verified fragments of RISC-V machine code as needed), or it can be pretty-printed to C and fed to a traditional C compiler.

3.3 The Anatomy of a Rupicola Lemma

Rupicola has two compilation judgments and accordingly two kinds of compilation lemmas: one to generate Bedrock2 statements and one for expressions. Here we focus on the former; the latter is described in a case study in section 4.

We write the judgment for statements as a Hoare triple $\{t; m; l; \sigma\}$ c $\{P\ p\}$. In the precondition t is the trace accumulated up to this point in the program, m the memory, l the locals, and σ the environment of functions that the program may call; in the postcondition P is a predicate (which will become useful once we compile monadic programs), and p is a Gallina value (the source program); and c is the Bedrock2 program being derived (always an evar). The judgment states that running c with the given starting state (precondition) leads to a final state verifying $P\ p$ (here P is partially applied; the result is a predicate on trace, memory, and locals).

As an example, here is a statement lemma about turning Vector.put v i b, the replacement at index i with value b in a length-indexed vector v of bytes (sometimes written as v[i ← b]), into store (v_var + I) B, the Bedrock2 version of a pointer assignment.

The lemma has 5 premises. Below, the first one indicates that local variable v_var contains a pointer v_ptr, and the second one states that memory m contains vector v at address v_ptr, alongside some separately framed memory r:

```
Lemma compile_vector_put {n} t m l σ...:
  map.get l v_var = Some v_ptr →
  (vector_value v_ptr v ∗ r) m →
```

The third and fourth premises are expression-compilation subgoals expressing the fact that I and B evaluate to i and b (by convention in this lemma we write deeply embedded terms in uppercase):

```
  EXPR m l I (of_bounded_nat i) →
  EXPR m l B (of_byte b) →
```

The final premise is a statement-compilation subgoal that generates a program K implementing the rest of (k a') of the original computation, using the mutated vector.[11]

```
  (∀ m',
    let v' := Vector.put v i b in
    (vector_value v_ptr v' ∗ r) m' →
    { t; m'; l; σ } K { pred (k v') }) →
```

Finally, we have the lemma's conclusion, which directly relates the two programs and their continuations:

```
  { t; m; l; σ }
    seq (store (var + I) B) K
  { pred (let/n v' as v_var :=
          Vector.put v i b in k v') }.
```

---

[11]Most Rupicola lemmas include such continuations; this is more convenient than using a generic sequencing lemma (a cut) because Bedrock2's predicates are asymmetric (the postcondition is a predicate, whereas the precondition is a collection of values that are universally quantified over in the Coq context of the proof).

3.4 Rupicola's Architecture

Rupicola is divided into a minimal core (definitions, notations, forward-reasoning tactics, and supporting architecture) and a collection of extensions. Section 4 covers some of these extensions; here we focus on two particularly interesting parts of Rupicola's code generation.

3.4.1 Compiling Effectful Programs.

Rupicola's source programs are pure, but leveraging the target language's native effects is crucial to getting good performance. In Rupicola, effects are classified into two categories: intensional and extensional.

Intensional effects are not explicitly encoded in the source (they do not appear in type signatures). Instead, they are introduced by special-casing certain code patterns through compiler extensions.

State and certain aspects of allocation are handled this way in Rupicola. For state, in particular, we do not typically use an explicit encoding: instead, we add lemmas to map e.g. list accesses to pointer dereferences, or pure replacements in a list to pointer assignments. Allocation of short-lived objects on the stack is handled similarly; we discuss it in a case study in subsubsection 4.1.2.

In general, intensional effects are either inferred or introduced explicitly by adding semantically transparent annotations to source programs (that is, annotations that do not change the meaning of the program). For example, every let-binding in functional models fed to Rupicola is annotated with the name of the variable it binds, allowing the compiler to decide when to mutate an object and when to allocate a new one based on the user's choice of names (in general, Rupicola expects input programs to be sequences of let-bindings, one per desired assignment in the target language). Similarly, to indicate that a let-binding should result in a copy instead of a mutation, a user might wrap the value being bound in a call to a copy function of type $\forall \alpha.\alpha \to \alpha$. Finally, while in simple cases data-structure mappings can be inferred automatically, in complex cases the user can control memory layout explicitly by using modules that transparently wrap underlying functional types (for example, the ListArray module reexposes list operations but tells Rupicola to use a contiguous array).

With this lightweight approach to intensional effects, and especially mutation, compiled programs can make full use of low-level state while source programs remain easy to reason about, with no explicit heap at the source level. This is a key advantage of Rupicola's intensional encoding of effects: it essentially does not impede verification efforts. When proving a functional model against a higher-level specification, annotations can simply be unfolded away: Rupicola's name-carrying let-bindings unfold to regular let-bindings, functions like copy above simply disappear, and modules wrapping standard types unfold to reveal them.

Extensional effects, in contrast, are introduced using explicit monadic encodings: users start with a pure specification, implement a functional model of it using monads, and then compile that model with Rupicola. This is how Rupicola handles nondeterminism and I/O, for example.

Rupicola's compilation judgment is phrased in a way that allows lemmas about nonmonadic terms to apply regardless of the source program's ambient monad: when compiling a pure binding in a monadic computation (bind (return a) k), the shape of the simplified term (let x := a in k x) allows us to apply any lemma that supports a. This means that Rupicola has, for example, a single lemma for compiling (pure) addition, applicable to all monadic programs.

Lemmas about monadic computations are a bit trickier. Recall the vector-compilation lemma from subsection 3.3: its conclusion was of the form $P(\texttt{let}/\texttt{n v} := \texttt{Vector.put} \ldots \texttt{in}$ k v), and the following compilation goal was simply P (k v). In the monadic case, we might similarly expect to start from P (bind ma k) and obtain as our next compilation goal P (k a)… but for which a? For $P : \texttt{M A} \rightarrow \texttt{state} \rightarrow \texttt{Prop}$ and a term bind ma k, we need to find a relation between P (bind ma k) *st* and P (k a) *st* for all *st* and for some (potentially universally quantified) value a. We guarantee that this relation exists by restricting P and requiring it to be formulated in terms of a monad-specific *lift* so that the postcondition always has shape lift P (bind ma k).

For the nondeterminism monad, for example, we encode a nondeterministic computation returning a value of type A as A → Prop (for example, a list of n unspecified natural numbers is represented as (λ l ⇒ length l = n)). Then, we require predicates to be lifted using the function $P \mapsto \lambda \, ma \, st. \, \exists a, \, ma \, a \wedge P \, a \, st$, which is such that $\{t; m; l; \sigma\}$ c {lift P (bind ma k)} is implied for all *a* by ma *a* $\wedge$ $\{t; m; l; \sigma\}$ c {lift P (k a)} (this is similar to what happens with nonmonadic bindings presented in subsection 3.3, but the value is now constrained by the computation ma).

For the writer monad, we encode a computation as a pair of a value and some accumulated output. Then, we require that predicates be lifted using the function lift := $o \, P \mapsto \lambda \, ma \, st. \, P$ (fst ma) (o ⧺ snd ma) *st*. Parameter o of the lift accumulates previous output, allowing us to compile monadic binds by accumulating their output into that parameter while reducing the source term. Here the relation is that $\{t; m; l; \sigma\}$ c {lift o P (bind ma k)} is implied by $\{t; m; l; \sigma\}$ c {lift (o ⧺ snd ma) P (k (fst ma))}.

3.4.2 Predicate Inference for Conditionals and Loops.

One of the hardest parts of conventional automated verification is inference of invariants at control-flow join points. Loop invariants are the classic tricky example, though a reduced version of the problem arises with conditionals. Rupicola has a leg up over classic approaches, in that the "specification" to be proved is just a functional program that we

may mention directly in invariants. As a result, we have fully automatic and predictable generation of loop invariants that merely need to characterize the connection between functional models and mutable low-level state.

To understand why compiling loops and conditionals poses specific challenges in Rupicola, consider a simple example: suppose that we are compiling code that writes value x to a memory cell at address p conditionally on a test t and returns a Boolean indicating whether a write has happened (in code: let r, c := (if t then (true, put c x) else (false, c)) in k c, a trivial compare-and-swap, with k standing for the program's continuation). We start with locals {"c": p} (variable "c" contains pointer p) and a memory predicate cell p c (stating that cell c is in a block of memory at address p). The lemma that we will use to compile this conditional will have a premise corresponding to the compilation of k. What will that premise look like? We might naively attempt to compile both branches and then merge their strongest postconditions. The result, unfortunately, is a new predicate (t $\wedge$ cell p (put c x)) $\vee$ ($\neg$t $\wedge$ cell p c) that is incomprehensible to later compilation steps: code in k will refer to the new value of c, and accordingly the compiler will look for a fact of the form cell ?p (if t then … else …)) — not a disjunction.

In other words, Rupicola's compilation frequently matches (syntactically) against a logical context that captures the state reached after symbolically executing the already-derived prefix of the output program. Because of this, we need precise control over the shape of the facts that are learnt as compilation progresses.

Therefore, instead of naive strongest postconditions, we apply the following heuristic to find an invariant.

1. Identify targets of the control-flow construct (loop or conditional) based on the names in the corresponding bindings. In the compare-and-swap example above, this would be two variables, "r" and "c".
2. For each target, determine whether it is a scalar or a pointer by inspecting the current locals and memory predicate. In the CAS example, we would determine that "r" is a scalar and "c" is a pointer: "r" because we do not find a binding for it in the map of locals, and "c" because the binding we find for it ("c": p) is to a pointer (p appears in the separation-logic predicate cell p c).
3. For each scalar, abstract over the corresponding binding in the locals. For each pointer, abstract over the corresponding entry in the predicate describing the memory. For CAS, we build a new map of locals {"c": p, "r": _} and a new memory predicate cell p _.
4. Close over the results. For CAS, we obtain the predicate (λ (r, c) l m ⇒ l = {"c": p, "r": r} $\wedge$ (cell p c) m).

The resulting predicate is a template parameterized on the values of the variables being created or mutated: to obtain a plain predicate, we need to supply concrete values. For

forward edges (conditionals) these values are exactly the (output of the) source programs being compiled: for CAS we obtain (λ l m $\Rightarrow$ let r, c := (if t then (true, put c x) else (false, c)) in (l = {"c": p, "r": r} $\land$ (cell p c) m)).

Loops are trickier to deal with: to reason about the body of a loop, we need to characterize the state of the program after some unknown number of iterations. In our case, this means instantiating the state representation predicate with symbolic variables that represent the program's state partway through iteration.

Classic verification depends on human-written loop invariants for this task. However, since our source programs are purely functional, we have a better option: we can use a complete characterization of the loop's behavior! For most programs, we can construct a symbolic representation of the program's state at iteration n by building a term corresponding to n steps of its execution (its first n iterations for numeric loops, or the first n elements of the input list for a map or fold, etc.). So, unlike a traditional loop-verification exercise in which we would have a human collect relevant properties into an invariant, we create a closed-form term parameterized by the (symbolic) iteration number and let users reason directly about it.[12]

As a concrete example, say we are compiling the loop let c := Nat.iter 10 incr c in k c (where incr increments the content of a cell, and Nat.iter n composes a function with itself n times). We obtain a general invariant parameterized over the loop-modified variables: (λ i l m $\Rightarrow$ let c := iter i incr c in l = {"c": p} $\land$ (cell p c) m), and hence we have access to a precise description of the state of c while compiling the loop's body.

We have loop-compilation lemmas for a wide variety of loops, each customized to provide optimally readable intermediate states. For example, our lemma that connects map to a for loop exposes intermediate states of the form map f (first n l) ++ (skip n l). Then, compiling loop bodies is like a classic Hoare-logic proof, where we know the invariant holds for iteration i and must prove it for iteration i+1. If we establish that connection, we are allowed to assume the invariant for iteration n, going into the code afterward. This process works without extensions for all examples presented in this paper.

The devil is in the details, though, of how we prove all the logical side conditions that arise during compilation of the body. We use two different approaches, depending on the kind of property:

Properties inherent to the choice of representation of a value (we call them *structural*) are encoded in separation-logic predicates. This is the case for properties like the length

of an object not changing when it is mutated, for example. Concretely, in our original uppercasing, we chose a separation-logic predicate that captured the length of the string in addition to its contents. Structural properties are automatically captured by our loop-invariant inference.

Properties specific to a particular algorithm or program (we call them *incidental*) are proven at the source level and recovered during compilation using hints. For example, if in addition to incrementing a cell our loop also accessed an array at the index corresponding to the value of the cell (arr[*p]), we would want to prove that after each iteration, the value in the cell is still within the bounds of the array. With our approach, rather than encoding them as low-level loop invariants, users prove incidental properties directly at the source level, by proving theorems about partial executions of their loops (iteration over part of a list or a range of numbers). For example, a user may prove that for all i, get (iter i incr c) equals get c + i. Plugging this as a compilation hint would then allow a linear solver to prove side conditions like 0 $\leq$ get (iter i incr c) $\leq$ length arr from preconditions about c and length arr.

4 Evaluation

We claim that Rupicola's novelty is its combination of extensibility, foundational proofs, and performance. The first and third claims are measurable. To support them, we evaluated Rupicola from three angles: programmer experience, expressivity, and performance. For the first two we used case studies; for the last we used benchmarks.

4.1 Programmer Experience and Expressivity

4.1.1 Extensibility. For Rupicola to generate code whose performance matches that of handwritten programs, we need users to be able to plug in new translation strategies, new hints, and new rewrites — relational compilation is the key to making this work, and it shines in particular when compared to traditional compilers and rewriting-based approaches.

In our experience, developing new programs in Rupicola often requires extensions, but these extensions are almost always very simple to plug in, assuming reasonable familiarity with our framework. Table 1 summarizes some examples of estimated incremental effort.

Table 1. Incremental verification effort for user extensions, in lines of Coq code. Development times are rough estimates.

Domain	Operation	Lemma	Proof	Time
nondet	alloc, peek	26+24	17+11	13+6 min
cells	get, put	22+23	5+ 3	7+3 min
	iadd	31	7	8 min
io	read, write	25+26	7+10	11+8 min

---

[12]Crucially, the predicate template captures all low-level details, and the symbolic instantiation is in terms of the functional source program.

Adding support for new monads is also straightforward, though naturally a bit more complicated. As a concrete example, we estimate that adding support for a writer monad starting from a blank file required about an hour and a half, with a bit over 15 minutes spent defining the monad and proving its properties (17 lines of code, 5 lines of proofs), 30 minutes spent setting up the compilation of that monad (56 lines of code, 8 lines of proofs), 20 minutes to add a Gallina primitive and compilation lemmas for it (mapping writes to I/O trace operations at the Bedrock2 level; 50 lines of code, 15 lines of proofs), 15 minutes to write a small example and compile it (4 lines of Gallina model, 6 lines for the Bedrock2 signature, and 1 line for the compilation "proof": compile.), and about 3 seconds to derive the actual code (compiler performance is discussed in more detail in subsection 4.3). The same example written by imitating other monad examples would probably take roughly a third to half as long.

4.1.2 Case Study: Implementing Compiler Extensions to Support New Low-Level Patterns. We implemented two useful extensions that expose different memory-management features of the target language.

Stack allocation. Bedrock2 supports (lexically scoped) stack allocations: a block of code can be wrapped in a binding construct giving it access to a pointer to a block of compile-time constant-size memory allocated on the stack. This is particularly useful for any program that needs access to a small working area, and unlike a global buffer it does not pollute external specifications (beyond changing the function's stack-space requirements, which Bedrock2 tracks). We added support for two new source constructs. For programs that immediately initialize their stack-allocated objects, we added a special identity function stack. When Rupicola sees `let` x := stack (term) `in` ..., it generates a stack allocation in Bedrock2 and resumes compilation with the plain program `let` x := term `in` .... Another form is for objects that are not initialized and must be modeled as beginning with non-deterministic contents. However, we proved a compilation lemma that applies when the resulting compilation is still provably deterministic (independent of initial bytes in the stack region). The implementation costs for these features were very similar: about 20 to 30 lines of lemmas and 5 to 10 lines of proofs, plus about 20 lines of typeclasses definitions and instances.

Inline tables. Inline tables are another Bedrock2 feature that is usefully exposed at the functional level; they are const arrays local to a Bedrock2 function, useful for implementing lookup and translation tables. The Gallina API that we implemented is exactly the same as that for arrays, except that only one operation (`get`) is available. Crucially, the API does not impede reasoning about the code: simply unfolding the definition of `InlineTable.get` reveals that it is just the function nth on lists. The API is complicated by polymorphism over types of values stored in a table. We did have to write hundreds of lines of proof to support reading full 32-bit words from tables, as opposed to tens of lines for reading bytes; but most of the additional effort was from proving properties that should be part of Bedrock2 itself.

4.1.3 Case Studies: Compiler Development and End-to-End Proofs. This section attempts to give a sense of the effort involved in developing and using relational compilers. Both case studies are developed in much greater detail in chapters 5.1.3 and 5.1.4 of [39].

Rupicola's expression compiler. Rupicola is really two relational compilers rolled into one: one targeting Bedrock2's statements and one targeting its expressions. Originally, however, we assumed that the expression part of the compilation process was so simple that it would not warrant the cost of relational compilation. Instead, we compiled expressions by reifying them into an AST type and then using a very simple verified compiler targeting Bedrock2's expression language, and we expected to handle all necessary extensions by plugging in new cases in our reflection tactics and proofs.

This was a miscalculation: extending that compiler was complicated (it required modifications in increasingly complex Coq tactics), and customizing its output for a specific program required duplicating the entire compiler to change just one case. Eventually we switched to relational compilation. The code went down from 450 lines to about 250 lines, and extending it was so smooth that we were soon back to about 400 — but now with support for machine words, bytes, Booleans, integers, two representations of natural numbers, and expressions with casts between different types. The overall impact on compilation times was reasonable: less than 30% overall.

End-to-end verification. Narrowly speaking, how to produce code suitable for compilation with Rupicola is out-of-scope for this paper: program synthesis, automated refinement, interactive refinement, and manual programming-and-proving are all reasonable approaches. Still, expressivity is a concern, so for multiple of our example programs we wrote end-to-end proofs connecting abstract specifications to Rupicola's inputs. We start from specifications capturing the desired behavior with no concerns of performance or even executability. Then we write annotated functional models, which we verify (by hand) against the original specifications. Finally, we find and add missing compilation lemmas interactively by running the compiler and inspecting compilation failures, if any. This process produces Bedrock2 code, which can either be compiled to RISC-V, yielding an end-to-end proof from high-level specifications to assembly, or translated to C and run through a traditional compiler. Because all reasoning happens on shallowly embedded programs, the verification experience is one that interactive theorem

provers excel at: proving equivalences between relatively small pure functions that operate on inductive data types. As a result, the required effort was generally low (minutes for trivial cases to hours for more complex ones).

Chapter 5.1.4 of [39] presents details about a concrete example (the IP-checksum program benchmarked below).

4.2 Benchmarks

Flexibility and extensibility are not the only metrics that we are optimizing for: they are in service of generating code that competes with handwritten C on performance.

To support our performance claims, we took a collection of tasks for which existing C implementations were available, implemented corresponding programs in Coq, and used Rupicola to compile them. Here, we give evidence that the performance of the resulting code is on par with hand written C programs. To run these programs we do not use Bedrock2's compiler to RISC-V; instead we use a simple pretty-printer to C to feed our programs to a regular C compiler (it would be possible to use Bedrock2's compiler or CompCert for greater assurance, albeit at a performance cost).

We chose programs from a variety of domains, including string manipulation, hashing, and packet-manipulating (network) programs. Not discussed in the following is an additional suite of dozens of programs testing features around arithmetic, monadic extensions, and stack allocation (a subset of which are covered in Table 1).

Table 2 gives a short description of each program that we benchmarked, and Figure 2 shows the results of benchmarking (running on an Intel Core i5-1135G7 @ 2.40GHz). As usual, benchmarks involving C compilers are very sensitive to small encoding decisions, so we measure performance across three compilers: overall the differences both in favor and against Rupicola are within the expected fluctuations across optimizing compilers, though we do suffer from a missed vectorization opportunity in upstr with GCC.

For space reasons we do not include detailed benchmarks of the OCaml code that Coq's built-in extraction can generate; suffice to say that we have we have found it to perform multiple orders of magnitude slower than the C code generated by Rupicola, even after tweaking Coq's extraction to produce more efficient code.[13] It is possible to improve the performance of the OCaml code further using potentially unsound extraction commands, but only up to a point; and each new customization of the extraction process is one more opportunity for subtle bugs.

Additional discussion of these performance results, including a detailed analysis of performance discrepancies, is presented in chapter 5.2.1 of [39].

---

[13]In many of these examples it does not even make sense to give a speedup ratio between Rupicola and code exported by Coq, because Rupicola changes the asymptotic complexity of the code it extracts (e.g. by changing a linear nth-element lookup to a constant-time pointer dereference).

Table 2. Our benchmark suite. "Source", "Lemmas", and "Hints" measure programmer effort in lines to write the original program and its signature, to prove the properties needed by Rupicola to compile it, and to construct the compiler, respectively. "End-to-End" indicates whether we have proofs from high-level specifications. The remaining columns describe which compiler extensions each program leverages.

| Name	Source	Lemmas	Hints	End-to-End	Arithmetic	Inline	Arrays	Loops	Mutation
fnv1a	35	-	2		✓		✓	✓	
Fowler-Noll-Vo (noncryptographic) hash									
utf8	56	-	6	✓	✓	✓			
Branchless UTF-8 decoding									
upstr	21	-	6	✓	✓		✓	✓	✓
In-place string uppercase (Box 1)									
m3s	11	-	-		✓				
Scramble part of the Murmur3 algorithm									
ip	37	3	7	✓	✓		✓	✓	
IP (one's-complement) checksum (RFC 1071)									
fasta	19	6	5	✓		✓	✓	✓	✓
In-place DNA sequence complement									
crc32	31	16	3		✓	✓	✓	✓	
Error-detecting code (cyclic redundancy check)									
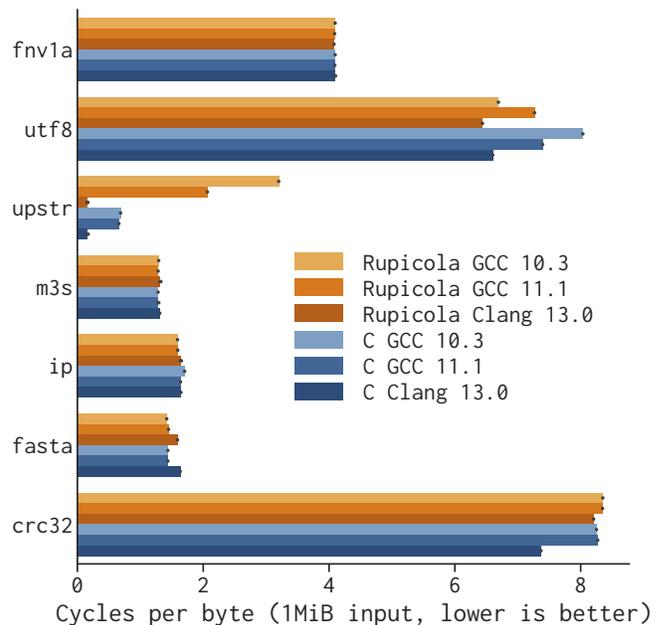
Figure 2. Performance benchmarks: Rupicola vs. handwritten C. Error bars indicate 95% confidence intervals for the mean over 1000 runs. Discrepancies are typically due to missed vectorization opportunities [39].

4.3 Limitations

In this section we try to summarize aspects that limit Rupicola's applicability.

Scalability and compilation times. While the programs that Rupicola produces are fast, Rupicola itself is not: it runs at the speed of Coq's proof engine, which in our experience means compiling anywhere between 2 and 15 statements per second. We are primarily focused on small programs (tens to hundreds of lines), so these issues have not proven particularly disruptive. They could be solved, we expect, with moderate amounts of engineering: most of them stem from API misalignments that force us to use inefficient workarounds (e.g. some of our derivations spend as much as 80% of their time in `autorewrite`, repeatedly failing to apply a handful of rewrites). Beyond this, Rupicola's intrinsic complexity should be essentially linear in the program size — plus time spent discharging logical side conditions, which may involve arbitrarily complex reasoning but in practice has not been an excessive part of our running time.

Expertise. To use Rupicola with its existing lemmas, users need general Coq knowledge (to write functions), a basic understanding of Rupicola's function specifications, and knowledge of what is available in Rupicola's standard library (data structures, algorithms, and annotations). It is possible to compile simple programs that way with no knowledge of Rupicola's internals. To extend Rupicola (by writing new lemmas), users need some familiarity with its compilation process, plus enough experience to write and verify snippets of Bedrock2 code (compilation lemmas connect Gallina to Bedrock2, so new lemmas require new Bedrock2 proofs). Ltac experience is also required when lemmas have side conditions not handled by Rupicola's existing automation.

Expressivity. Some low-level code patterns do not map naturally to purely functional models and hence are hard to generate with Rupicola. A prime example of this is nonlocal control flow: while patterns like exceptions (using the error monad) or early exits from loops are relatively easy to support in Rupicola, code that uses arbitrary `goto`s or `longjmp` to implement complex control flow does not lend itself nicely to generation with Rupicola.

Trusted base. All code written in Rupicola comes with proofs of (total) correctness, but there are still moving parts and potential sources of bugs. Briefly, those are: Rupicola's inputs (user errors in specifications); Coq's proof checker; the environment in which programs are generated and run; and the unverified parts of Bedrock2's compilation toolchain (when pretty-printing to C).

In a traditional pipeline, like Coq's extraction to OCaml, users have to trust (1) the correctness of their own customizations of Coq's native extraction (textual replacements at the OCaml level), as well as (2) Coq's extraction machinery itself (a complex program spanning a few thousands lines), and (3) the downstream (OCaml) compiler. In Rupicola (1) is eliminated, since all extensions are verified; (2) is almost entirely eliminated, because pretty printing to C is done by a very small program of just 200 lines that is essentially implementing an identity function (so there is hope that it may eventually be reasonably robust); and (3) remains (the C compiler), though it could be eliminated by using a verified C compiler — at a performance cost.[14]

5 Related Work

Rupicola draws inspiration from, but shares no code with, our own previous work, Fiat-to-Facade (F2F) [41]. It was the first demonstration of an end-to-end pipeline for deriving code automatically from high-level specifications to low-level code, and it strove for both performance and extensibility in a foundational context. Unfortunately, it also suffered from issues that eventually convinced us to restart from scratch: F2F's linear target language caused us performance issues; it only proved partial, not total correctness; it used setoid rewriting, leading to compilation runs that took minutes, not seconds; it used tactic hooks to build compilers, leading to much less extensibility; and it hardcoded the nondeterminism monad. Rupicola solves all these issues by developing the simple and nicely compositional framework of relational compilation.

Also closely related to this work is Imperative/HOL [17, 18]. Early work targeted a shallowly embedded language with GC, but the latest work extracts directly to LLVM. The main difference is the scope of the translation: LLVM/HOL uses a direct embedding of LLVM into HOL, so a form of relational compilation is used to perform what is essentially a one-to-one translation where all effects in the source are encoded extensionally. Rupicola, on the other hand, accepts more complex inputs and supports most effects intensionally.[15] Another closely related line of work uses proof-producing extraction to translate HOL programs to deeply embedded CakeML (a dialect of ML for which there exists a verified compiler) [11, 16, 29, 30]. It bridges a much narrower gap than Rupicola does (it targets a language with garbage collection), but in exchange it offers a much more complete translation pipeline, in the sense that it supports a better-defined and larger part of its input language, HOL.

Members of the F* team take a slightly different approach in KreMLin [45], an extraction framework from *Low** (an imperative subset of F*) to C: the extraction process is not

---

[14]Of course, if a performance penalty is acceptable, then using Bedrock2's own *verified* compiler to RISC-V eliminates both (2) the pretty-printer and (3) the unverified downstream compiler.

[15]Another difference is that Rupicola integrates into a verified pipeline: code extracted with Rupicola can be soundly compiled and linked against other code written in Bedrock2 or directly in machine code, within Coq, whereas there is no verified implementation of LLVM in Isabelle/HOL today.

verified (though a proof-producing strategy for a subset was described [25]), but the close match between the Low* style and C means that KreMLin's trusted base is reasonably small, and the emphasis on output-code quality means that C code from KreMLin can easily be integrated into larger, potentially unverified C programs (as has in fact been done with cryptography routines [44]). This strategy is viable because F* provides convenient facilities for reasoning about stateful programs in shallowly embedded style, making it possible to prove the connection between code written in high-level functional style and low-level imperative style without resorting to reasoning about deeply embedded low-level terms. More recent developments explore metaprogramming and code generation using stateful functors [43].

The authors of Cogent [33] take a different approach. Instead of translating between two languages (one functional and one imperative), they guarantee (using a restrictive type system) that all valid Cogent programs admit efficient implementations that do not depend on a runtime or a garbage collector (similar to the way in which Facade [41] was essentially a linear type system on top of Cito [56]). As a result, unlike Rupicola and F2F, Cogent is complete, but it is also much more restrictive: it does not support arbitrary user-supplied extensions, nor custom translation of specific high-level patterns; and all optimizations must be expressed in the source program, not as transformations to be applied as part of the source-to-target translation process.

Coq extraction and verified compilation. Coq's traditional extraction mechanism [36, 37] is not machine-verified, but it is proven on paper [21], and it supports a form of (unsound) extension by remapping constructors and functions to arbitrary OCaml expressions, a feature very commonly used in large extracted Coq developments. With sufficiently arcane combinations of extraction commands, it is often possible to improve performance significantly, at some risk to soundness. More principled are approaches based on reification: with a sufficiently restricted subset of Gallina, it is possible to reify terms into a deeply embedded AST using Ltac's reflection and certify correctness of that translation by interpreting deeply embedded results back into Gallina [27, 58].

CertiCoq [1] is a verified compiler from Coq to assembly: it starts by reifying Gallina into a deeply embedded AST and then proceeds as a traditional verified compiler. Unlike in Rupicola, the extraction process is not extensible, so users pay the price of inefficiencies at the Gallina level; but, in exchange, the compiler is complete: it supports all of Gallina. More generally, the last few years have seen an explosion of work on the topic of compiler verification, most notably with CakeML [16] and CompCert [20]. F2F depended on a verified compiler called Cito [56]; Rupicola uses Bedrock2 [8].

Translation validation. Complete verification of a compiler can be onerous, and verifying that the compiler produces correct outputs on all inputs is often qualitatively more complex than establishing that property for any given input/output pair. As a result, many verified systems employ translation validation instead of verification: a (trustworthy, ideally verified) checker is used to confirm, for each run of the compiler, that the outputs are correct. The problem is undecidable for most input and output languages, so a variety of heuristics coexist in the literature [14, 32, 42, 52–54, 57], some quite close to the relational-extraction style that we advocate [9]. It would not be unreasonable to classify Rupicola as a translation-validation system, since it uses unverified Ltac scripts to generate output programs along with "witnesses" of correctness in the form of Coq proof terms.

Other compilation, optimization, and synthesis work. Many recent developments seek to reduce the cost of running functional programs, e.g. in OCaml [4] or Coq [22, 34]. Related work has also explored dynamically discovering the sort of mutation that Rupicola introduces statically [7, 46, 47, 55]. Earlier work on decompilation into logic handles user extension similarly to Rupicola, albeit with code generation separate from translation validation [28, 31]. Stepping back further, Rupicola's design shares a lot with work on extensible compilation and domain-specific languages for optimization [19, 35, 49, 51] and more generally with work on automated program derivation and program synthesis, all the way back to deductive program synthesis [24].

6 Conclusion

We have introduced the framework of *relational compilation* and presented Rupicola, a relational-compilation toolkit that leverages modular compiler extensions to derive high-performance, verified low-level programs automatically from functional sources. Rupicola is unique in its combination of extensibility, foundational proofs, and performance. We are in the process of extending it to support further application domains, and we are looking into integrating its verified outputs into existing widely used libraries.[16]

Acknowledgments

---

[16]Readers curious to learn more about relational compilation and about Rupicola are encouraged to consult the first author's PhD dissertation [39].

References

[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A Verified Compiler for Coq. In *3rd International Workshop on Coq for PL (CoqPL 2017)*.

[2] Andrew W. Appel. 2011. Verified Software Toolchain. In *20th European Conference on Programming Languages and Systems* (Saarbrücken, Germany) *(ESOP 2011)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

[3] Edwin C. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[4] Pierre Chambart, Mark Shinwell, Damien Doligez, and OCaml Contributors. 2016. *Optimization with FLambda*. https://ocaml.org/manual/flambda.html

[5] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *25th International Conference on Automated Deduction (CADE 2015)*. 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[6] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. Association for Computing Machinery, 689–700. https://doi.org/10.1145/2676726.2677006

[7] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. 1994. OPAL: Design and Implementation of an Algebraic Programming Language. In *International Conference on Programming Languages and System Architectures*. 228–244. https://doi.org/10.1007/3-540-57840-4_34

[8] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 604–619. https://doi.org/10.1145/3453483.3454065

[9] Yannick Forster and Fabian Kunze. 2019. A certifying extraction with time bounds from Coq to call-by-value λ-calculus. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 17:1–17:19.

[10] Konstantin Romanov Gabriel Scherer, Frédéric Bour. 2020. *TRMC, reloaded*. https://github.com/ocaml/ocaml/pull/9760

[11] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. 2018. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. In *9th International Joint Conference on Automated Reasoning (IJCAR 2018)*. Springer International Publishing, 646–662. https://doi.org/10.1007/978-3-319-94205-6_42

[12] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *3rd ACM SIGPLAN History of Programming Languages Conference (HOPL 2007)*. 1–55. https://doi.org/10.1145/1238844.1238856

[13] Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In *27th European Symposium on Programming (ESOP 2018)*. 999–1026. https://doi.org/10.1007/978-3-319-89884-1_35

[14] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416.

[15] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB. In *9th International Conference on Interactive Theorem Proving (ITP 2018)*. Springer, 362–369. https://doi.org/10.1007/978-3-319-94821-8_21

[16] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. 179–192. https://doi.org/10.1145/2535838.2535841

[17] Peter Lammich. 2015. Refinement to Imperative/HOL. In *6th International Conference on Interactive Theorem Proving (ITP 2015)*. 253–269. https://doi.org/10.1007/978-3-319-22102-1_17

[18] Peter Lammich. 2019. Generating Verified LLVM from Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. 22:1–22:19. https://doi.org/10.4230/LIPIcs.ITP.2019.22

[19] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. 2005. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. 364–377. https://doi.org/10.1145/1040305.1040335

[20] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*. 42–54. https://doi.org/10.1145/1111037.1111042

[21] Pierre Letouzey. 2002. A New Extraction for Coq, In 2nd International Workshop on Types for Proofs and Programs. *Types for Proofs and Programs*, 200–219. https://doi.org/10.1007/3-540-39185-1_12

[22] John M. Li and Andrew W. Appel. 2021. Deriving Efficient Program Transformations from Rewrite Rules. *Proceedings of the ACM on Programming Languages* 5, ICFP, Article 74 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473579

[23] Andreas Lööw and Magnus O. Myreen. 2019. A proof-producing translator for Verilog development in HOL. In *7th International Workshop on Formal Methods in Software Engineering (FormaliSE@ICSE 2019)*. 99–108. https://doi.org/10.1109/FormaliSE.2019.00020

[24] Zohar Manna and Richard J. Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems* 2, 1 (Jan. 1980), 90–121. https://doi.org/10.1145/357084.357090

[25] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European Symposium on Programming (ESOP 2019)*. Springer International Publishing, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

[26] Henry Massalin. 1987. Superoptimizer - A Look at the Smallest Program. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1987)*. 122–126. https://doi.org/10.1145/36177.36194

[27] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 172–185. https://doi.org/10.1145/3167089

[28] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2008. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *8th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008)*. 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.24

[29] Magnus O. Myreen and Scott Owens. 2012. Proof-producing Synthesis of ML from Higher-order Logic. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*. 115–126. https://doi.org/10.1145/2364527.2364545

[30] Magnus O. Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming* 24, 2-3 (Jan. 2014), 284–315. https://doi.org/10.1017/s0956796813000282

[31] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. 2009. Extensible Proof-Producing Compilation. In *18th International Conference on Compiler Construction (CC 2009)*. 2–16. https://doi.org/10.1007/978-

3-642-00722-4_2

[32] George C. Necula. 2000. Translation validation for an optimizing compiler. *ACM SIGPLAN 2000 conference on Programming language design and implementation - PLDI '00* (2000). https://doi.org/10.1145/349299.349314

[33] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021), 25. https://doi.org/10.1017/S095679682100023X

[34] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional Optimizations for CertiCoq. *Proceedings of the ACM on Programming Languages* 5, ICFP, Article 86 (Aug. 2021), 30 pages. https://doi.org/10.1145/3473591

[35] Lionel Parreaux. 2020. *Type-Safe Metaprogramming and Compilation Techniques For Designing Efficient Systems in High-Level Languages.* Ph. D. Dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-10285

[36] Christine Paulin-Mohring. 1989. *Extraction de programmes dans le Calcul des Constructions.* Theses. Université Paris-Diderot - Paris VII. https://tel.archives-ouvertes.fr/tel-00431825

[37] Christine Paulin-Mohring and Benjamin Werner. 1993. Synthesis of ML Programs in the System Coq. *Journal of Symbolic Computation* 15, 5-6 (May 1993), 607–640. https://doi.org/10.1016/S0747-7171(06)80007-6

[38] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah E. Chasins, and Rastislav Bodik. 2014. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*. 396–407. https://doi.org/10.1145/2594291.2594339

[39] Clément Pit-Claudel. 2022. *Relational compilation: Functional-to-imperative code generation for performance-critical applications.* Theses. Massachusetts Institute of Technology. https://pit-claudel.fr/clement/PhD/RelationalCompilation_Pit-Claudel_2022.pdf

[40] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. *Artifact for Rupicola paper at PLDI 2022.* https://doi.org/10.5281/zenodo.6330740

[41] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *10th International Joint Conference on Automated Reasoning (IJCAR 2020, Vol. 12167)*. Springer International Publishing, 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

[42] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

[43] Jonathan Protzenko and Son Ho. 2021. Zero-cost meta-programmed stateful functors in F*. *CoRR* abs/2102.01644 (2021). arXiv:2102.01644 https://arxiv.org/abs/2102.01644

[44] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, and et al. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. *2020 IEEE Symposium on Security and Privacy (SP)* (May 2020). https://doi.org/10.1109/sp40000.2020.00114

[45] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-level Programming Embedded in F*. *Proceedings of the ACM on Programming Languages* 1,

ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

[46] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage free reference counting with reuse. *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (June 2021). https://doi.org/10.1145/3453483.3454032

[47] Wolfram Schulte and Wolfgang Grieskamp. 1991. Generating Efficient Portable Code for a Strict Applicative Language. In *PHOENIX Seminar and Workshop on Declarative Programming*. 239–252. https://doi.org/10.1007/978-1-4471-3794-8_16

[48] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. 256–270. https://doi.org/10.1145/2837614.2837655

[49] Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010)*. 111–121. https://doi.org/10.1145/1806596.1806611

[50] The Coq Development Team. 2021. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.4501022

[51] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*. 132–141. https://doi.org/10.1145/1993498.1993514

[52] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI 2011)*. Association for Computing Machinery, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

[53] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL 2008)*. Association for Computing Machinery, New York, NY, USA, 17–27. https://doi.org/10.1145/1328438.1328444

[54] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI 2009)*. Association for Computing Machinery, New York, NY, USA, 316–326. https://doi.org/10.1145/1542476.1542512

[55] Marian Vittek. 1996. A Compiler for Nondeterministic Term Rewriting Systems. In *7th International Conference on Rewriting Techniques and Applications (RTA 1996)*. 154–167. https://doi.org/10.1007/3-540-61464-8_50

[56] Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler Verification Meets Cross-language Linking via Data Abstraction. In *2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*. 675–690. https://doi.org/10.1145/2660193.2660201

[57] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proceedings of the ACM on Programming Languages* 5, ICFP, Article 84 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473589

[58] Vadim Zaliva and Matthieu Sozeau. 2019. Reification of shallow-embedded DSLs in Coq with automated verification. In *5th International Workshop on Coq for PL (CoqPL 2019)*.